

Group Mutual Exclusion in Linear Time and Space

Yuan He and Krishnan Gopalakrishnan

Department of Computer Science

East Carolina University

Greenville, NC -27858

Abstract—Group Mutual Exclusion (GME), introduced by Joung, is a natural generalization of the classical mutual exclusion problem. In GME, when a process leaves the remainder section it requests a “session”. Processes are allowed to be in the critical section at the same time if they have requested the same session. We present an algorithm for the GME problem that satisfies the properties of Mutual Exclusion, Starvation Freedom, Bounded Exit, Concurrent Entry and FCFS. Our algorithm has $O(N)$ RMR (Remote Memory Reference) Complexity, $O(N)$ Shared Space Complexity. Moreover it uses only bounded shared variables and only simple read and write operations.

Jayanti et al. presented an algorithm satisfying the same set of properties. Their algorithm has $O(N)$ Shared Space Complexity and is considered to have $O(N)$ RMR (Remote Memory Reference) Complexity in the CC Model. Their algorithm is based on an earlier algorithm by Hadzilacos whose shared space complexity is $O(N^2)$ and whose RMR Complexity was claimed to be $O(N)$ in the CC Model. Both of these algorithms are based on an elegant single bit classical mutual exclusion algorithm developed independently by Burns and Lamport. We show that this algorithm is of RMR complexity $\Theta(N^2)$ in the CC Model and thus demonstrate that neither the algorithm of Jayanti et al. nor that of Hadzilacos is of complexity $O(N)$. As far as we know, our algorithm is the first linear time (RMR) and linear (shared) space algorithm for the GME Problem that uses only bounded shared variables and only simple read and write operations. Moreover, our algorithm has $O(N)$ RMR complexity in both the CC Model and the DSM Model.

We also present a simpler algorithm that has linear time and space complexity, but uses unbounded shared variables. This algorithm is developed by generalizing the well-known Lamport’s Bakery Algorithm for classical mutual exclusion problem. Although an earlier paper by Takamura and Igarashi also generalizes Lamport’s Bakery Algorithm to GME, their algorithm does not have concurrent entry and bounded exit properties.

I. INTRODUCTION

Mutual Exclusion is a classical problem in distributed computing introduced by Dijkstra in 1965 [3]. In this problem, processes repeatedly cycle through four sections of code viz., Remainder Section, Entry Section, Critical Section and Exit Section, in that order. The problem consists of designing code for the Entry Section and the Exit Section such that the mutual exclusion property is satisfied. Mutual exclusion property states that no two processes can be in the critical section at the same time. To ensure liveness, it is often required to satisfy the property of *Starvation Freedom*. Starvation freedom property states that if no process stays in the critical section forever, then any process which enters the Entry Section eventually enters the critical section. To ensure fairness, it is often required to satisfy the property of *First Come First*

Served (FCFS). Informally, the FCFS property requires that processes are allowed into the critical section in the same order in which requests are made.

The *Group Mutual Exclusion (GME)* problem, introduced by Joung in 1998 [8], is a natural generalization of the classical mutual exclusion problem. In GME, when a process leaves the remainder section, it requests a “session” and wants to enter into the critical section. Unlike the classical mutual exclusion problem, multiple processes are allowed to be in the critical section at the same time, provided they have requested the same session. As before, we can think of the processes repeatedly cycling through four sections of code viz., Remainder Section, Entry Section, Critical Section and Exit Section, in that order. An execution of the last three sections will be called an *attempt*. The process picks a session number when it leaves the remainder section and this session number can be different in each attempt. A process is said to be an *active* process, if it is in one of its attempts. We will say that two active processes are in *conflict* if their session numbers are different.

A solution to the GME problem consists of developing code for the Entry Section and the Exit Section so that the following four properties are satisfied.

- P1 **Mutual Exclusion:** No two conflicting processes can be in the Critical Section at the same time.
- P2 **Starvation Freedom:** If no process stays in the critical section forever, then any process that enters the Entry Section eventually enters the Critical Section.
- P3 **Bounded Exit:** After entering the Exit Section, a process is guaranteed to leave it within a bounded number of its own steps.
- P4 **Concurrent Entry:** In the absence of conflicting processes, a process in the Entry Section should be guaranteed to enter the Critical Section within a bounded number of its own steps.

The *Concurrent Entry* property is crucial to the GME problem. It was stated informally by Joung [8] and then was later formalized by Hadzilacos [5]. If we do not care to ensure Concurrent Entry property, then almost any algorithm for classical mutual exclusion problem also solves the GME problem. The intent of this property is to ensure concurrency; active processes that request the same session, in the absence of conflicting processes, should be allowed to enter the CS without unnecessary synchronization among themselves.

In this paper, we will be exclusively concerned with fair solutions to GME. A fair solution to GME will be required

to satisfy the *First Come First Served (FCFS)* property in addition to the abovementioned four properties. Informally, FCFS property demands that conflicting processes be allowed into the critical section in the same order in which they made requests. In order to formalize this idea, we will think of the Entry Section as though it is made up of two sections viz., Doorway Section and Waiting Section. The Doorway Section is a wait-free section of code, i.e., a section of code that can be completed by a process in a bounded number of its own steps. The Waiting Section is where the actual synchronization with other conflicting processes occurs and may entail indefinite waiting.

P5 FCFS If process i completes the doorway before process j enters the doorway and the two processes request different sessions, then j does not enter the critical section before i .

The notion of doorway was originally introduced by Lamport [9] for classical mutual exclusion.

A. Model

We consider a system consisting of N processes, named $1, 2 \dots N$ and a set of shared variables. Each process also has its own private variables. Processes can communicate only by writing into and reading from shared variables. An execution is modeled as a sequence of process steps. In each step, a process performs some local computation or writes into a shared variable or reads from a shared variable. The processes take steps asynchronously. Specifically, this means that an unbounded number of steps of some other process could be performed in between two consecutive steps of a process. We assume that our processes are *live*; this means that if a process has not terminated it will eventually execute its next step.

We allow only simple read and write operations on shared variables. Although we assume that these read and write operations are atomic, we do not assume that processes have access to more powerful synchronization operations such as atomic test-and-set, compare-and-swap etc.

There are two general architectural paradigms considered for shared memory in the literature, viz., Distributed Shared Memory (DSM) Model and Cache-Coherent (CC) Model. In the DSM model, each processor has its own memory module and each shared variable is assigned to a particular processor. When a process is referencing a shared variable, it is locally accessible, if it is allocated to that processor itself; on the other hand it is a remote access, if it is allocated to some other processor. In a CC machine, each processor has a private cache and some hardware protocol ensures cache coherence i.e., all copies of the same variable in different local caches are consistent. On such a machine, a shared variable becomes locally accessible by migrating it to a local cache. We assume that once a shared variable is brought into local cache, it remains there until it is invalidated as a result of it being modified by some other processor.

By the term time complexity, we mean Remote Memory Reference (RMR) complexity in this paper. This is because remote memory references are the most time consuming

operations as they involve interconnect traversal. By the term space complexity, we mean that total amount of shared space a solution entails. We do not count the private variables when measuring space complexity.

We consider two different situations, one in which the shared variables are bounded registers and one in which the shared variables are unbounded registers. Note that bounded registers mean there is a bound on the maximum value that can be stored in the register. Unbounded registers can store arbitrarily large values in them.

B. Our Contribution and Related Work

Joung's original algorithm for the GME problem satisfies the four basic properties. It does not satisfy the FCFS property. Moreover, it has unbounded RMR Complexity.

Later, Hadzilacos [5] gave a nice solution for the GME problem that also has the FCFS property. Their algorithm can be thought of as a modular composition of two independent algorithms, one, "the FCFS algorithm" provides FCFS property (but not necessarily guarantee mutual exclusion) and the other, "the ME algorithm" provides ME property (but not necessarily FCFS). Their algorithm has shared space complexity of $O(N^2)$. It was claimed that the algorithm has RMR complexity of $O(N)$ in the CC Model. The algorithm was using only bounded shared variables and simple read and write operations on them. It was left as an open problem to develop a fair solution (satisfying P1 through P5) for the GME problem that runs in linear time and space.

Couple of years later, Jayanti et al. [6] presented an algorithm as a solution to the abovementioned open problem. They retained the idea of modular composition and also the "ME algorithm" that Hadzilacos used. They came up with a clever modification to the "FCFS algorithm" of Hadzilacos to reduce the space complexity. Although they did not explicitly claim so, their algorithm is considered to be of linear space and time. For example, the recent paper by Bhatt and Huang [1] explicitly states that the RMR Complexity of the algorithm by Jayanti et al. is $O(N)$.

Both of the above algorithms use an algorithm developed independently by Burns [2] and Lamport [10] as the "ME algorithm". This is an elegant algorithm that uses just one bit of shared space per process. In Section II we show that this algorithm actually has an intricate structure and has the worst case RMR complexity of $O(N^2)$. It follows that algorithms of both Hadzilacos and Jayanti et al. for solving GME are of RMR complexity $O(N^2)$. Hence, the open problem posed by Hadzilacos still remains open. In Section III, we present a fair algorithm (satisfying P1 through P5) for the GME problem that runs in linear time and space. Our algorithm uses only bounded shared variables and uses only simple read and write operations on them. To the best of our knowledge, our algorithm is the first one to solve the open problem originally posed by Hadzilacos.

Our algorithm is fairly complicated. If we are allowed to use unbounded registers, we present a simple algorithm to solve the GME problem (satisfying P1 through P5) that runs in

linear time and space. This simple algorithm is a generalization of the Classic Lamport's Bakery Algorithm. To the best of our knowledge, even under these conditions no other linear time and space algorithm for GME problem exists in the literature. Although Hadzilacos mentions a simpler version of his algorithm in [5] that uses $O(N)$ space, if unbounded registers are allowed, that also suffers an RMR Complexity of $O(N^2)$ as it uses the same "ME algorithm".

Takamura and Igarashi [13] have also developed an algorithm for the GME problem by generalizing the idea behind Lamport's Bakery Algorithm for the classical mutual exclusion algorithm. However, their algorithm does not have bounded exit and concurrent entry properties (P3 and P4) and so it is not really a valid solution.

Finally we mention that $O(\log N)$ RMR Complexity Algorithm has been developed by Bhatt and Huang in [1]. However, their algorithm uses LL/SC instructions which are powerful synchronization operations. In contrast, our algorithm uses only simple read and write operations.

II. A FLAW IN THE LITERATURE

The algorithm of Hadzilacos [5] as well as that of Jayanti et al. [6] can be viewed as a modular composition of an "FCFS Algorithm" and an "ME Algorithm". Moreover, they both use the same "ME Algorithm". Hadzilacos claims that his algorithm is of $O(N)$ RMR Complexity in the CC Model. The algorithm of Jayanti et al. is also considered to be of $O(N)$ RMR Complexity in view of the claim by Hadzilacos. We show in this section that the "ME Algorithm" used by them are of complexity $O(N^2)$ in the CC Model thus invalidating their claims.

Assume that processes i and j are two active processes requesting different sessions. As per the FCFS property, if process i completes the doorway, before process j enters the doorway, then j does not enter the CS before i . In conjunction with the starvation freedom property, this means that i will enter the CS before j . However, it is possible that while process i is in the middle of its doorway, process j might enter its doorway. Under these circumstances, we will say that processes i and j are doorway-concurrent. If two processes are doorway-concurrent, then the FCFS property does not dictate as to who should get into the CS first. In fact, both processes can get out of the "FCFS algorithm" and can potentially enter the CS simultaneously. In order to prevent this from happening, we need the "ME Algorithm". In the worst case, consider a situation where all the n processes are doorway-concurrent and are requesting different sessions. Then all of them can get out of the "FCFS Algorithm" and move on the "ME Algorithm".

In order to simplify the presentation from now on in this section, we consider the "ME algorithm" alone. This elegant "ME algorithm" is independently discovered by Burns [2] and Lamport [10]. This algorithm is depicted in Figure 1. In this algorithm *Competing* is a shared array of size N . Each element of the array is a boolean variables, initialized to false.

Fig. 1. Burns-Lamport ME algorithm

```

1: L: Competing[ $i$ ] = true
2: for  $j = 1$  to  $i - 1$  do
3:   if Competing[ $j$ ] then
4:     Competing[ $i$ ] = false
5:     wait until not Competing[ $j$ ]
6:     goto L
7:   end if
8: end for
9: for  $j = i + 1$  to  $N$  do
10:  Wait until not Competing[ $j$ ]
11: end for

```

The code given in Figure 1 is for process i . j is a private variable.

We now briefly describe the algorithm. Every process i owns a single bit *Competing*[i]. Only process i can write into *Competing*[i]; Other processes can read it. Before entering the CS, process i sets its bit to true and checks all lower numbered processes (lines 1-3). If any of them, say j , is found to have set its bit to true, then process i resets its bit to false, allowing the smaller-numbered process to make progress. It then waits for j 's bit to become false and then restarts the competition to enter the CS by going to line 1. Having checked all lower-numbered processes, i then checks the higher-numbered ones and waits for each of them to set its bit to false. Now, however, while i is waiting it does not set its bit to false. After that i enters the CS. It turns out that this simple algorithm guarantees mutual exclusion. We refer the reader to [10] (see also [2]) for a proof of correctness.

It is easy to see that the algorithm is of unbounded RMR Complexity in the DSM Model. This is because the busy-wait loops in lines 5 and 10 are accessing shared variables that are not allocated to processor i and hence this is not a local-spin algorithm.

To analyze the RMR complexity under the CC Model, consider the following sequence of events

- 1) Process N sets its bit to true.
- 2) Process $N - 1$ sets its bit to true.
- 3) Process N checks all lower-numbered processes and finds that Process $(N - 1)$'s bit is set. So, Process N sets its bit to false and waits for *Competing*[$N - 1$] to become false.
- 4) Process $N - 2$ sets its bit to true.
- 5) Process $N - 1$ checks all lower-numbered processes and finds that Process $(N - 2)$'s bit is set. So, Process $N - 1$ sets its bit to false and waits for *Competing*[$N - 2$] to become false.
- 6) Process N now finds *Competing*[$N - 1$] to be false and so restarts the competition by setting its bit to true.
- 7) Process N checks all lower-numbered processes and finds that Process $(N - 2)$'s bit is set. So, Process N sets its bit to false and waits for *Competing*[$N - 2$] to become false.
- 8) Process $N - 3$ sets its bit to true.

- 9) Process $N - 2$ checks all lower-numbered processes and finds that Process $(N - 3)$'s bit is set. So, Process $N - 2$ sets its bit to false and waits for Competing[N-3] to become false.
- 10) Process N now finds Competing[N-2] to be false and so restarts the competition by setting its bit to true.
- 11) Process N checks all lower-numbered processes and finds that Process $(N - 3)$'s bit is set. So, Process N sets its bit to false and waits for Competing[N-3] to become false.
- 12) .
- 13) .
- 14) .
- 15) Process N checks all lower-numbered processes and finds that Process 1's bit is set. So, Process N sets its bit to false and waits for Competing[1] to become false.
- 16) Process 1 checks all higher-numbered processes and finds that all the bits are false and enters CS.
- 17) Process 1 exits the CS and sets its bit to false.

Note that during the above sequence of events, process N got blocked by each one of the lower numbered processes once. At the end of the above sequence, process 1's request is satisfied. Also, at the end of the above sequence, the Competing bit of processes 2 through N are all false. Now, we can iterate the same sequence of events, but this time with only processes 2 through N participating. We can recursively iterate the same sequence of events again and again until finally we have only process N participating.

The net effect is that in this worst case scenario, process N got blocked by process $(N - 1)$ a total of $(N - 1)$ times, by process $(N - 2)$ a total of $(N - 2)$ times and so on. So, the total number of times, process N gets blocked by some other process is

$$\sum_{k=1}^{N-1} k = N(N - 1)/2 = O(N^2)$$

In the CC model, at least one remote memory reference is involved each time a process gets blocked and hence the worst case RMR complexity of the algorithm in Figure 1 is $O(N^2)$ in the CC Model. It was erroneously claimed in [5] and tacitly inherited in [6] that this algorithm is of $O(N)$ RMR. That claim is clearly wrong as illustrated above. Hence the problem of developing a linear time (RMR) and linear (shared) space algorithm for the GME problem, originally posed by Hadzilacos, is still open. In the next section, we develop such an algorithm

III. GME IN LINEAR TIME AND SPACE

In this section, we present our linear time and linear space algorithm to solve the Group Mutual Exclusion Problem. The algorithm satisfies properties P1 through P5. It has an RMR Complexity of $O(N)$ in both the DSM Model and the CC Model and has a shared space complexity of $O(N)$. It uses only bounded shared variables and only simple read and write operations to manipulate the shared variables. Our algorithm

can also be thought of as a modular composition of an "FCFS algorithm" and an "ME algorithm". In fact, we will use the same FCFS algorithm that Jayanti et al. used in [6]. The crucial difference is in the "ME algorithm" where we use our own algorithm of $O(N)$ RMR Complexity. As we pointed out in Section II, the "ME algorithm" used by Hadzilacos in [5] and by Jayanti et al. in [6] are of $O(N^2)$ RMR complexity in CC Model and unbounded RMR complexity in the DSM Model.

We begin by presenting a two process mutual exclusion algorithm from the literature in subsection III-A. We then extend it naturally to an N process solution in subsection III-B and observe that this extension will blow up the space complexity to $O(N^2)$. We then show how to overcome these difficulties and cut down the space complexity to $O(N)$ by using Gödel encoding technique in subsection III-C. Finally we analyze the RMR Complexity of the resulting algorithm and show it to be $O(N)$ in subsection III-D. It is then a simple matter to put the "FCFS Algorithm" of Jayanti et al. and our "ME Algorithm" to form a linear time and linear space algorithm for the GME Problem.

A. A Solution for Two Process Mutual Exclusion

Yang and Anderson proposed a two process mutual exclusion algorithm in [14]. We present a quick review of this algorithm as this is the starting point for our design.

We denote the two processes as i and j . We use three shared variables in this algorithm. The first one is C , an array variable indexed by i or j . Each array element takes the value of either true or false and is initialized to false. The second one is S , an array variable indexed by i or j . Each array element takes the value of either 0, 1 or 2 and is initialized to 0. In case of DSM model, we allocate $C[i]$ and $S[i]$ to processor i and $C[j]$ and $S[j]$ to processor j . The third one is an ordinary variables T , which takes a value of i or j and the initial value does not matter. Also, it can be allocated to either one of the two processors in the case of DSM model. The code for process i is shown in Figure 2. The code for process j is symmetrical.

Variable $C[i]$ is set to true whenever process i is interested in entering its critical section and helps to inform process j of its intent to enter its critical section. It remains true until it gets access to the critical section, performs the critical section and exits the critical section (lines 2 through 15). Variable $C[j]$ is similarly used by process j . In the event that only one of the two processes wants to enter the critical section, it can freely do so. In the event that both processes want to enter the critical section at the same time, we use the value of the variable T to decide which process should be given priority. The processes are allowed to enter the critical section in the same order in which they updated the variable T . Hence, the variable T is called the tie-breaker variable. Note that T takes on one of two values i or j . Variable $S[i]$ is used by process i for busy-waiting. Variable $S[j]$ is used by process j similarly. Hence, busy-waiting is done by local spinning.

When process i wants to enter the critical section, it informs process j of its intention by setting $C[i]$ to be true. Then, process i assigns its identifier i to the tie-breaker variable T

Fig. 2. Yang-Anderson Algorithm: Two-process case

```

1:  $C[i] = \text{true};$ 
2:  $T = i;$ 
3:  $S[i] = 0;$ 
4: if  $C[j] \neq \text{false}$  then
5:   if  $T = i$  then
6:     if  $S[j] = 0$  then
7:        $S[j] = 1$ 
8:     end if
9:     Wait until  $S[i] \geq 1;$ 
10:    if  $T = i$  then
11:      Wait until  $S[i] = 2;$ 
12:    end if
13:  end if
14: end if
15: Critical Section;
16:  $C[i] = \text{false};$ 
17: if  $T \neq i$  then
18:    $S[j] = 2;$ 
19: end if

```

and initializes its spin location $S[i]$ to zero. If process j has not shown interest in entering the critical section, i.e., if $C[j]$ is false in line 4, then process i proceeds directly to its critical section. Otherwise, i reads the tiebreaker variable T . If $T \neq i$, which implies $T = j$, then i can enter the critical section as clearly i updated the tie-breaker variable T first.

If $T = i$ holds, then either process j executed line 2 before process i or process j has executed line 1, but not line 2. In the first case, process i should wait until process j exits the critical section, whereas, in the second case process i should be allowed to proceed to its critical section. We differentiate between these two situations, by making process i execute lines 6 through 13. Lines 6 and 7 are executed by process i to release process j , in the event that process j is waiting for i to update its tie-breaker variable. Line 9 makes process i wait until j had a chance to update the tie breaker variable. When process i reaches line 10, if $T = i$ holds, we know for sure that process j updated the tie-breaker variable first as we know that process j had a chance to update the tie-breaker variable in this attempt. Hence j should be given priority and so i waits until $S[i] = 2$ (a condition that will be made true by process j when it exits its critical section).

After executing the critical section, process i informs process j that it is finished by setting $C[i]$ to false. If $T = j$, in which case process j is trying to enter its critical section, process i updates $S[j]$ to 2 so that j does not have to wait any longer. It is easy to see that this is a constant RMR algorithm in both the DSM model and the CC model.

B. Natural Extension of the solution to N processes

Once, we have an algorithm for the two process mutual exclusion problem, it can easily be extended to solve the more general N process case. As observed by Petersen in [12], the two process solution can be used repeatedly by a process to

Fig. 3. Generalization to N -process case

```

1: for  $j = 1$  to  $N$  do
2:   if  $j \neq i$  then
3:     ENTRY( $i, j$ )
4:   end if
5: end for
6: Critical Section
7: for  $j = N$  down to 1 do
8:   if  $j \neq i$  then
9:     EXIT( $i, j$ )
10:  end if
11: end for

```

resolve possible conflict with each one of the other processes.

We will use ENTRY(i, j) to denote the entry section from the two-process solution that process i executes to compete with process j (the portion of the code starting from line 1 through line 14 in Figure 2). We will use EXIT(i, j) to denote the corresponding exit section (the portion of the code starting from line 16 through line 19 in Figure 2). Then, for the N -process case, we have the simple algorithm for process i shown in Figure 3.

It should be straightforward to see that the generalization is correct and it has $O(N)$ RMR Complexity as two-process version has $O(1)$ RMR Complexity.

However, the problem is that except for the $C[i]$ shared variable, the other shared variables cannot be reused to maintain correctness of the algorithm. As the processes run asynchronously, it is possible that while process i is competing with process j , some other process k may be competing with process i . Hence, the tie-breaker variable T and the spin variable $S[\cdot]$ really need to be different for each instance of competition between two processes. While it is not too difficult to rectify this problem, by using extra shared variables appropriately, it would increase the shared space complexity of the algorithm. In particular, we need one tie-breaker variable for each pair of processes and as there are $\binom{N}{2}$ different possible pairs of processes, we need $O(N^2)$ tie-breaker variables. Since, we are interested in building a linear time (RMR) and linear space (shared) algorithm, we need to overcome this difficulty.

C. Cutting down space complexity

In this section, we discuss how we overcome the difficulty and cut down the space complexity to $O(N)$ while at the same time maintaining the RMR complexity at $O(N)$.

As mentioned before a process i can signal its intent to enter the critical section to other processes by setting its $C[i]$ shared variable to true. When process i exits the critical section, it can set $C[i]$ to false. So, this involves only a total of $O(N)$ shared space. However, we need special techniques to deal with the S variable and the T variable.

First, we will describe as to how we deal with the spin variable S . We use a technique called **Gödel encoding** originally used by Gödel in the proof of his famous incompleteness

Fig. 4. Function Extract(i,j)

```

1:  $x = 0$ ;
2:  $y = G[i]$ ;
3: while  $y \bmod p_j = 0$  do
4:    $x = x + 1$ ;
5:    $y = y/p_j$ ;
6: end while
7: Return  $x$ 

```

theorem [4]. Let $p_1, p_2, p_3, \dots, p_N$ denote the first N prime numbers (i.e., $p_1 = 2, p_2 = 3$, and so on). Process i is allocated a shared variable $G[i]$. The value of this shared variable at any point of time will be

$$G[i] = p_1^{s_1} p_2^{s_2} \dots p_{i-1}^{s_{i-1}} p_{i+1}^{s_{i+1}} \dots p_N^{s_N},$$

where $s_1, s_2, s_3, \dots, s_N$ are the current values of the spin variables used by process i when competing with processes $1, 2, 3, \dots, N$ respectively. Essentially, $G[i]$ is the Gödel encoding of the sequence of values of the spin variables. We have an elementary function $extract(i, j)$, which returns the exponent of the prime number p_j from the Gödel encoding $G[i]$. The code for this function is shown in Figure 4. Thus essentially, we are able to simulate N different shared variables by just one variable using the Gödel encoding. Note that the spin variables take on a value of 0, 1 or 2 at any point of time and so none of the exponents will ever be larger than 2.

Next, we describe as to how we deal with the tie-breaker variable. Every process i maintains a variable $B[i]$. The value of this shared variable at any point of time will be

$$B[i] = p_{i+1}^{t_{i+1}} p_{i+2}^{t_{i+2}} \dots p_N^{t_N},$$

where p_{i+1} is the $(i+1)^{th}$ prime number, p_{i+2} is the $(i+2)^{th}$ prime number and so on. Here the exponents, $t_{i+1}, t_{i+2}, \dots, t_N$ take on a value of either 0 or 1 at any point of time. Suppose the processes i and j are competing with each other and i is less than j , then the tie-breaker variable is encoded in $B[i]$. If the tie-breaker variable has a value of i , then we set the exponent t_j of p_j in $B[i]$ to be 0 and on the other hand if the tie-breaker variable has a value of j , then we set the exponent of t_j of p_j in $B[i]$ to be 1. Once again, with this encoding and interpretation, we are able to simulate $\binom{N}{2}$ shared tie-breaker variables using only $O(N)$ shared variables.

The resulting algorithm has three shared array variables C, G and B each of size N . Hence, the shared space complexity of the algorithm is clearly $O(N)$. We summarize the meaning of each variable now. $C[i]$ is a boolean variable and is used to signal the intent of process i to enter the critical section. $G[i]$ is an integer variable and is the Gödel encoding of all the spin variables that process i may need in different competitions. $B[i]$ is an integer variable and is used to encode the tie-breaker variables of all competitions in which process i competes with a process having larger index. The C array is initialized to false. The G and B array are initialized to 1.

Our algorithm is presented in Figure 5. In line 1, process i is declaring its intention to enter CS and in line 2, we are

setting all our spin locations to 0 by initializing $G[i]$ to 1. In lines 3 through 5, we are preparing to compete with each other process that is also interested in entering the critical section. Lines 6 through 16 are basically setting the tie breaker variable to i . Lines 17 through 21 are indicating to the process j that we are done with setting our tie-breaker variable. In line 22, we wait until process j had a chance to set the tie-breaker variable. In lines 23 through 24, we wait for process j to exits its critical section, in case the other process j has priority.

After exiting the critical section, we set $C[i]$ to false in line 30 to signal other processes that we are done with the critical section. In lines 31 through 43, we release each one of the other processes, in case it was waiting for process i to leave the critical section.

D. Analysis of the Solution

Let us now analyze the RMR Complexity of the algorithm. We first focus on the exit section. In lines 32 through 42, as there are no loops involved, there will only be constant number of remote memory references. As these lines are enclosed in a for loop that runs at most N times, the RMR complexity of the exit section is $O(N)$.

We now focus on the entry section. In lines 4 through 27, loops are involved only in the two *Await* statements of lines 22 and 24. So, apart from lines 22 and 24, there can only be constant number of remote memory references involved in lines 4 through 27. As these lines are enclosed in a for loop that runs at most N times, the RMR complexity of this portion of the code is also $O(N)$.

So, all that remains to be argued is that the total RMR Complexity of executing the await statements in lines 22 and 24 is also $O(N)$. The shared variable that is being accessed in these lines is the variable $G[i]$. As we are working under the CC model, when we first access this variable, we will bring it to the cache memory and keep using it, until the hardware protocol informs us that the value of the variable is invalidated. In that case, we will bring the new value again to the cache costing another RMR. So, the total number of remote memory references is basically a count of how many times the variable $G[i]$ can change while process i is waiting in the lines 22 and 24. It is important to realize that $G[i]$ can only increase during these wait statements. $G[i]$ starts with an initial value of 1. It can be increased by each other process j at most twice, once to indicate that j is done setting its tie breaker variable and once to indicate that j is done with the critical section. So, the total number of times the value of $G[i]$ can change while process i is waiting in lines 22 and 24 is $2(N-1)$. Hence, the overall RMR Complexity of the algorithm is $O(N)$.

Note that we are not arguing that the value changes to $G[i]$ in each pass through the lines 22 and 24 is constant. In fact, that is not necessarily true. Despite that fact, we are able to argue correctly that the total number of value changes to $G[i]$ in all the $N-1$ passes through lines 22 and 24 is $2(N-1)$.

Thus our mutual exclusion algorithm is a linear time (i.e., it has RMR Complexity of $O(N)$) and linear space (i.e., total amount of shared space complexity is $O(N)$) algorithm.

Moreover, our algorithm's RMR complexity is $O(N)$ even in the DSM Model, as the only shared variable being accessed in the busy-wait loops in lines 22 and 24 is $G[i]$ which is assigned to processor i in the DSM Model. The FCFS Algorithm of Jayanti et al. (see Figure 1 in [6]) is a linear time and linear space algorithm. Now, it is a simple matter to obtain a linear time and linear space algorithm for the Group Mutual Exclusion Problem by doing a modular composition of the "FCFS Algorithm" of Jayanti et al. and our "ME Algorithm". Considering the simplicity of doing so, for the sake of brevity, we omit a full description of the complete algorithm. The only point worth mentioning is that we will not compete with a process that requests the same session as ours in the GME algorithm.

IV. GENERALIZING LAMPORT'S BAKERY ALGORITHM

In this section, we present a very simple algorithm for the group mutual exclusion problem that has $O(N)$ RMR Complexity and $O(N)$ Shared Space Complexity. This algorithm also satisfies the properties P1 through P5 and uses only simple read and write operations to manipulate the shared variables. However, unlike our previous algorithm this one will use unbounded shared variables.

This algorithm is a generalization of Lamport's bakery algorithm (see [9]) for the classical mutual exclusion problem. It is based on the method commonly used in bakeries, in which a customer receives a token number upon entering the store and the holder of the lowest token number is the next one served.

This algorithm uses three shared variables. The first one is *Session*, an integer array of size N and $Session[i]$ indicates the session number that process i requests in this attempt to enter the critical section. The second one is *Token*, an integer array of size N and $Token[i]$ represents the token number selected by process i . The third one is *Choosing*, a boolean array of size N and $Choosing[i]$ is true would indicate that process i is currently attempting to determine its token number. The *Session* array and the *Token* array are initialized to zero and the *Choosing* array is initialized to false. It is easy to see that this algorithm is of shared complexity $O(N)$. However, the token numbers used by this algorithm will grow in an unbounded manner.

The code for process i is shown in Figure 6. When a process leaves the remainder section, it first selects its session number and places it in $Session[i]$. We assume that all session numbers are positive integers. Process i then sets its $Choosing[i]$ variable to be true to signal to other processes that it is currently attempting to select a token number. It selects its token number to be one more than the maximum of the token numbers of all other processes and places it in $Token[i]$. It then sets $Choosing[i]$ to false to signal to other processes that it is done with selecting its token number.

For each other process j , process i checks to see if it is also interested in entering the critical section (line 9). If it is not, then there is no problem; on the other hand, if it is also interested, it waits until the process j has selected

Fig. 5. Linear Time and Space ME Algorithm

```

1:  $C[i] = true;$ 
2:  $G[i] = 1;$ 
3: for  $j = 1$  to  $N$  do
4:   if  $j \neq i$  then
5:     if  $C[j] \neq false$  then
6:       if  $i < j$  then
7:          $k = i;$ 
8:         if  $B[k] \bmod p_j = 0$  then
9:            $B[k] = B[k]/p_j;$ 
10:        end if
11:      else
12:         $k = j;$ 
13:        if  $B[k] \bmod p_i \neq 0$  then
14:           $B[k] = B[k] \times p_i$ 
15:        end if
16:      end if
17:      if  $((k = i) \wedge ((B[k] \bmod p_j) \neq 0)) \vee$ 
18:         $((k = j) \wedge ((B[k] \bmod p_i) = 0))$  then
19:        if  $extract(j,i) = 0$  then
20:           $G[j] = G[j] \times p_i;$ 
21:        end if
22:      end if
23:      if  $((k = i) \wedge ((B[k] \bmod p_j) \neq 0)) \vee$ 
24:         $((k = j) \wedge ((B[k] \bmod p_i) = 0))$  then
25:        if  $extract(i,j) = 0$  then
26:           $G[i] = G[i] \times p_j;$ 
27:        end if
28:      end if
29:      end if
30:      end if
31:      end if
32:      end if
33:      end if
34:      end if
35:      end if
36:      end if
37:      end if
38:      end if
39:      end if
40:      end if
41:      end if
42:      end if
43:      end if

```

its token number in case it is a conflicting process. If j is indeed a conflicting process, then i waits until either j has exited the critical section or it has lower token number. It is easy to see that i enters the critical section if it has the smallest token number among all conflicting processes. This observation immediately implies that the algorithm has the FCFS property and the mutual exclusion property. It is trivial to notice that the algorithm has bounded exit property as the exit section is made up of two simple statements. It is not too difficult to see that this algorithm satisfies the concurrent entry property. This algorithm is also deadlock free as there cannot be a circular wait among the processes. Any algorithm that is both deadlock free and has FCFS property, also has starvation freedom property and so we can conclude that this algorithm has starvation freedom property. Thus this algorithm has all the five desired properties P1 through P5. For the sake of brevity, we defer a more formal presentation of the proof.

It is possible that two different processes may pick the same token number and in that we use the process identifier to resolve the ties. The relation “less than” on ordered pairs of integers is defined by $(a, b) < (c, d)$ if $a < c$ or if $(a = c)$ and $b < d$. This is used in line 12 of the algorithm.

This algorithm has unbounded RMR complexity under the DSM model as the waiting is not done using local spin variable. However, under the CC model, we will show that the RMR complexity is $O(N)$. In lines 8 through 15, there are only two loops viz., the busy-wait loops in line 10 and line 12. So, there can be only be a constant number of remote memory references in other lines. In line 12, when process i is busy waiting $Token[i]$ cannot change. If $Token[j]$ changes, it will either change to 0 or to a number higher than $Token[i]$. In either case, it will cause the wait loop to terminate. Hence, line 12 can only involve a maximum of three remote memory references (one for $Token[i]$ and two for $Token[j]$). Similarly in line 10, if $Choosing[j]$ changes, the new value will be false and so the loop will terminate. However, if $Session[j]$ changes, it could still be different from *mysession*. But, in this case, $Choosing[j]$ has to change to false before the session number can change again. So, line 10 can only involve a maximum of four remote memory references. So, there are only constant number of remote memory references in lines 8 through 15. As lines 8 through 15 is enclosed within a for loop that can run a maximum of $N - 1$ times, it follows that the entire entry section is of $O(N)$ RMR complexity. Note that lines 1 through 6 involves only constant number of remote memory references, except for the implicit loop in line 5. The implicit loop in line 5 has $O(N)$ RMR Complexity as it involves inspecting the token numbers of all other processes. Finally, it is also see that the exit section consisting of lines 18 and 19 involve exactly two remote memory references. Hence, the overall RMR complexity of this algorithm is $O(N)$.

An earlier paper by Takamura and Igarashi [13] also made an attempt to generalize Lamport’s Bakery Algorithm to Group Mutual Exclusion Problem. They present three different algorithms in that paper. Their first algorithm does not satisfy the starvation freedom property. Their second and third

Fig. 6. Generalization of Lamport’s Bakery Algorithm

```

1: while true do
2:   Remainder Section
3:    $Session[i] = mysession;$ 
4:    $Choosing[i] = true;$ 
5:    $Token[i] = 1 + \text{max of other token numbers};$ 
6:    $Choosing[i] = false;$ 
7:   for  $j = 1$  to  $N$  do
8:     if  $j \neq i$  then
9:       if  $Session[j] \neq 0$  then
10:        Wait until  $((Choosing[j] = false) \vee$ 
11:           $(Session[j] = mysession))$ 
12:        if  $Session[j] \neq mysession$  then
13:          Wait until  $((Token[j] = 0) \vee ((Token[i], i) <$ 
14:             $(Token[j], j)))$ 
15:        end if
16:      end if
17:    end for
18:    Critical Section
19:     $Session[i] = 0;$ 
20:     $Token[i] = 0;$ 
21: end while

```

algorithms, apart from being quite complicated, do not satisfy the concurrent entry property and bounded exit property. Their second and third algorithms however satisfy a weaker property known as *concurrent occupancy* in the literature. To the best of our knowledge, our algorithm shown in Fig. 6 is the most simple and elegant generalization of Lamport’s Bakery Algorithm for the Group Mutual Exclusion Problem.

V. CONCLUSION

In [5] Hadzilacos presented an algorithm for the Group Mutual Exclusion problem. The overall structure of this algorithm can be thought of as a modular composition of an “FCFS algorithm” and an “ME algorithm”. The shared space complexity of his algorithm is $O(N^2)$ and he claimed his algorithm to be of RMR Complexity $O(N)$ under the CC Model. He left it as an open problem to design a linear (RMR) time and linear (shared) space algorithm for the GME problem.

In [6], Jayanti et al. developed a different “FCFS algorithm” and used it to produce a solution for the GME problem by composing it with the same “ME algorithm” that Hadzilacos used. The shared space complexity of the algorithm by Jayanti et al. is $O(N)$. Although they do not explicitly claim in their paper, there appears to be a tacit assumption that their algorithm is of $O(N)$ RMR complexity. For example, in [1], they explicitly state that the RMR Complexity of the algorithm by Jayanti et al. is $O(N)$.

Both Hadzilacos and Jayanti et al. use an elegant algorithm developed independently by Lamport [10] and Burns [2] for providing mutual exclusion. We showed in this paper that this algorithm is really of RMR complexity $O(N^2)$ under the CC Model. Hence the algorithms of both Hadzilacos and Jayanti et

al. are also of RMR complexity $O(N^2)$. So, the open problem posed by Hadzilacos is not really solved by Jayanti et al.

We solve this open problem by presenting a linear time (RMR) and linear shared space algorithm to solve the GME Problem. Our algorithm satisfies the properties of mutual exclusion, starvation freedom, concurrent entry, bounded exit and FCFS (P1 through P5). Moreover, our algorithm uses only simple read and write operations to manipulate the shared variables. Also, all the variables used in our algorithm are bounded variables. To the best of our knowledge our algorithm is the first one having all of the above desirable properties. Finally, our algorithm is of RMR Complexity $O(N)$ under both DSM model and CC model as it uses local spin in all the busy-wait loops. In contrast, the algorithms of both Hadzilacos and Jayanti et al. are of unbounded RMR Complexity under the DSM Model and $O(N^2)$ RMR Complexity under the CC Model.

Our algorithm uses Gödel encoding technique in a novel manner to encode multiple pieces of data into a single variable and this helped in cutting down the space complexity. We believe that this technique is of independent interest and might be useful in other distributed algorithms as well.

One of the disadvantages of our algorithm is that even though all the shared variables are bounded variables, the bound is not independent of N . In particular, the variable $G[i]$ could become quite large as the value stored in it could be as big as the square of the product of the first N prime numbers. As a consequence of the prime number theorem [11], one gets an asymptotic expression for the N^{th} prime p_N as $p_N = O(N \ln N)$. Using this result, we can estimate the shared space complexity at the bit level, instead of at the register level, to be $O(N \log N)$. We leave it as an open problem to design an algorithm to solve the GME problem that has $O(N)$ shared space bit complexity in addition to having all the other properties that our algorithm had.

We also present a very simple and elegant generalization of Lamport's Bakery Algorithm to the GME Problem. Our solution satisfies all the five properties P1 through P5. This algorithm is of shared space complexity $O(N)$. It has unbounded RMR Complexity under the DSM Model and has $O(N)$ RMR Complexity under the CC Model. However, it has the limitation of using unbounded registers as shared variables.

Lamport's Bakery Algorithm for the classical mutual exclusion problem also has the limitation of using unbounded registers. However, Jayanti et al. were able to remove that limitation in [7]. We leave it as an open problem to investigate whether similar techniques could be used to overcome the same limitation of our generalization of Lamport's Bakery Algorithm to the GME Problem.

REFERENCES

- [1] V. Bhatt, C-C. Huang, Group Mutual Exclusion in $O(\log n)$ RMR, In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, (2010), 45—54.
- [2] James E. Burns, Complexity of Communication Among Asynchronous Parallel Processes, *Ph.D. Thesis, Georgia Institute of Technology*, (1981).
- [3] E. Dijkstra, Solution of a problem in concurrent programming control, *Communications of the ACM*, **8(9)** (1965), 569.
- [4] K. Gödel Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatsheft für Math. und Physik*, **38** (1931) 173-198.
- [5] V. Hadzilacos, A note on Group Mutual Exclusion, In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*, (2001), 100—106.
- [6] P. Jayanti, S. Petkovic and K. Tan, Fair Group Mutual Exclusion, In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, (2003), 278—284.
- [7] P. Jayanti, K. Tan, G. Friedland and A. Katz, Bounding Lamport's Bakery Algorithm, In *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'01)*, Lecture Notes in Computer Science, Springer Verlag, **2234** (2001), 261—270.
- [8] Y. Jeong, Asynchronous Group Mutual Exclusion, *Distributed Computing*, **13(4)** (2000), 189—206.
- [9] L. Lamport, A new solution of Dijkstra's concurrent programming problem, *Communications of the ACM*, **17(8)** (1974), 453—455.
- [10] L. Lamport, The mutual exclusion problem: Parts I and II, *Journal of the ACM*, **33(2)** (1986), 313—348.
- [11] A. Selberg, An elementary proof of the prime number theorem, *Annals of Mathematics*, **50(2)** (1949), 305—313.
- [12] G. L. Peterson, Myths about mutual exclusion problem, *Information Processing Letters*, **12(3)** (1981), 115-116.
- [13] M. Takamura and Y. Igarashi, Group Mutual Exclusion Algorithms Based on Ticket Orders, In *Proceedings of the 9th Annual International Computing and Combinatorics Conference (COCOON'03)*, Lecture Notes in Computer Science, Springer Verlag, **2697** (2003), 232—241.
- [14] J. H. Yang and J. Anderson, A fast, scalable mutual exclusion algorithm, *Distributed Computing*, **9(1)** (1995), 51—60.